Università degli Studi di Parma
Dipartimento di Ingegneria e Architettura
Sistemi operativi e in tempo reale – a.a. 2022/23

Cooperazione tra processi e thread mediante semafori

Cooperazione tra processi e thread concorrenti

- Due forme di cooperazione:
  - Scambio di un *segnale temporale* che indica il verificarsi di un dato evento
  - Scambio di *messaggi* generati da un thread e consumati da un altro

- La *cooperazione tra thread* prevede che la esecuzione di alcuni di essi *risulti condizionata* dall'informazione prodotta da altri
- La cooperazione implica *vincoli sull'ordinamento nel tempo* delle operazioni dei thread

Semafori e cooperazione

- Il meccanismo del semaforo, introdotto per risolvere problemi di competizione tra thread (mutua esclusione), consente di risolvere anche i problemi di cooperazione tra thread concorrenti
- La *precedenza* denotata dall'arco del grafo di precedenza può infatti essere realizzata con un *semaforo*

- → Rassegna di problemi di cooperazione di base e di altri problemi classici di sincronizzazione, risolti mediante semafori

Scambio di segnali

- *N* thread T1, T2, ... Tn attivati ad intervalli di tempo prefissati da un thread manager T0
- Vincoli:
  - l'esecuzione di Ti non può iniziare prima che sia giunto il segnale da T0
  - ad ogni segnale inviato da T0 deve corrispondere una attivazione di Ti

Scambio di segnali

- Ogni thread può essere regolato mediante un semaforo s_i con valore iniziale $s_{i0} = 0$

```
thread Ti:
   begin
        ...
        repeat
                wait(si);
                ...
        forever
   end;
```

```
thread T0:  //manager
   begin
        ...
        repeat
                ...
                signal(si);
                ...
        forever
   end;
```

Verifica di correttezza

- n1 = numero di richieste di attivazione da parte di Ti
  n2 = numero di segnali di attivazione inviati da T0
  n3 = numero di volte in cui Ti è stato attivato
- In ogni istante deve valere:

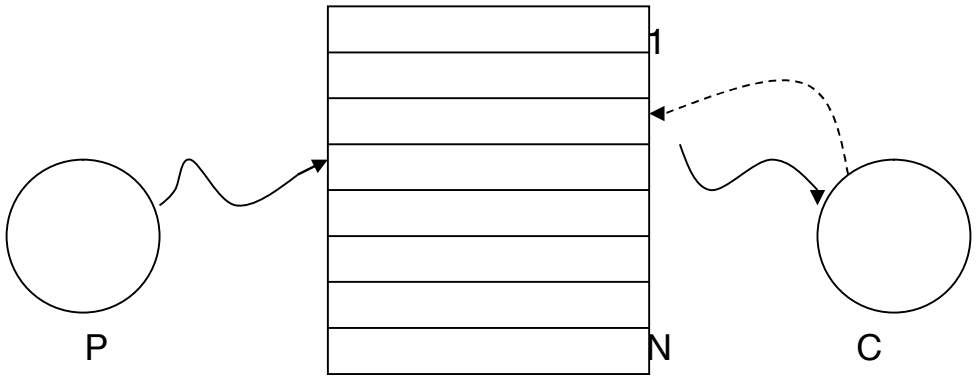
  (a) se $n2 \geq n1$ $n3 = n1$

  (b) se $n2 < n1$ $n3 = n2$
- La *relazione di invarianza* $nw(s) \leq ns(s) + s0$
  in questo caso diviene: $nw(s_i) \leq ns(s_i)$
  che coincide con le condizioni (a) e (b)

- Le *signal*(s_i) eseguite da T0 tengono traccia di tutte le richieste, che quindi non vengono perdute

Produttore-Consumatore

- Ipotesi di soluzione mediante *buffer limitato*: buffer in grado di contenere N messaggi, a cui accedono il processo P in scrittura ed il processo C in lettura



- Vincoli:
  - Il produttore *non può inserire* un messaggio nel buffer se questo è *pieno*
  - Il consumatore *non può prelevare* un messaggio dal buffer se questo è *vuoto*

Produttore-Consumatore

- Vincolo di correttezza della soluzione al problema mediante buffer limitato: $0 \leq d - e \leq N$
- Ove:

  d = numero dei messaggi depositati

  e = numero dei messaggi estratti

  N = dimensione del buffer

Produttore-Consumatore

- Schema iniziale come *processi asincroni*:

Produttore (P)	Consumatore (C)
repeat	*repeat*
<produzione messaggio>	<prelievo messaggio>
<deposito messaggio>	<consumazione messaggio>
forever	*forever*

- E' necessario introdurre una *sincronizzazione* tra i processi per evitare che P depositi messaggi sul buffer pieno o che C prelevi messaggi dal buffer vuoto

Produttore-Consumatore

- Introduciamo un semaforo *mess-disp* (v.i.: mess-disp = 0)

Produttore (P)

repeat
  <produzione messaggio>
  <deposito messaggio>
  signal(mess-disp)
forever

Consumatore (C)

repeat
  wait(mess-disp)
  <prelievo messaggio>
  <consumazione messaggio>
forever

- La soluzione evita l'*underflow* ma non l'*overflow*

Produttore-Consumatore

- La soluzione richiede due semafori:

  'messaggio disponibile' *mess-disp* v.i. = 0

  'spazio disponibile' *spazio-disp* v.i. = N

Produttore (P)

repeat

<produzione messaggio>

wait(spazio-disp)

<deposito messaggio>

signal(mess-disp)

forever

Consumatore (C)

repeat

wait(mess-disp)

<prelievo messaggio>

signal(spazio-disp)

<consumazione messaggio>

forever

Produttore-Consumatore

- E’ una soluzione *simmetrica*, non privilegia nessun processo
- P e C possono operare *in parallelo sul buffer, su messaggi diversi*
- Nell’ipotesi in cui i messaggi vengano prelevati dal buffer nel medesimo ordine in cui sono stati inseriti…
- P e C *non possono operare sul medesimo messaggio*, indipendentemente dalla sua lunghezza
  - P e C tentano di accedere allo stesso messaggio solo nelle condizioni limite di buffer pieno e buffer vuoto: in tali condizioni uno dei due processi è bloccato dalla wait

Verifica di correttezza

- La relazione $nw(s) \leq ns(s) + s_0$ diventa:

  (1) nw (spazio-disp) ≤ ns (spazio-disp) + N

  (2) nw (mess-disp) ≤ ns (mess-disp)

- In base all'ordine con cui vengono eseguite le primitive:

  (3) ns (mess-disp) ≤ d ≤ nw (spazio-disp) (produttore)

  (4) ns (spazio-disp) ≤ e ≤ nw (mess-disp) (consumatore)

- Da (3), (1), (4) si ha:

  d ≤ nw (spazio-disp) ≤ ns (spazio-disp) + N ≤ e + N

- Da (4), (2), (3) si ha:

  e ≤ nw (mess-disp) ≤ ns (mess-disp) ≤ d

- Da cui: e ≤ d ≤ e + N

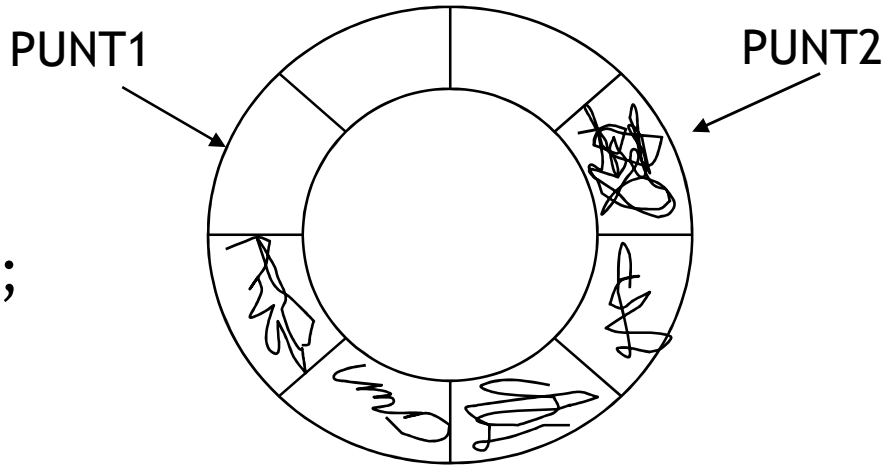
  0 ≤ d - e ≤ N

Produttore-Consumatore

- Produttore e consumatore non devono mai accedere contemporaneamente alla stessa posizione del buffer
- Possibile realizzazione mediante buffer circolare:

  PUNT1 = prima cella libera

  PUNT2 = ultima cella scritta



- Inserimento:

  buffer(PUNT1):=messaggio_prodotto;

  PUNT1:=(PUNT1+1) mod N;
- Prelievo:

  messaggio_prelevato:=buffer(PUNT2);

  PUNT2:=(PUNT2+1) mod N;

Produttore-Consumatore

- Si può avere interferenza solo con accesso concorrente di P e C e PUNT1=PUNT2
- PUNT1=PUNT2 può significare:
  - a) buffer tutto pieno → solo C può agire sul buffer
  - b) buffer tutto vuoto → solo P può agire sul buffer
- → la soluzione mediante semafori al problema Produttore-Consumatore è corretta anche nei casi a) e b)

Problemi Produttore-Consumatore

- Il problema presenta diverse varianti
- Abbiamo analizzato: problema del Produttore-Consumatore con buffer limitato, produzione e consumazione unitarie, consumazione nell'ordine di produzione
- Varianti:
  - Un-bounded buffer
  - Non unitary deposit and consumption
  - Un-ordered consumption
- Enjoy!

Produttore-Consumatore: esercizi /1/

Produttore (P)

```
repeat
    wait(spazio-disp)
    <produzione messaggio>
    <deposito messaggio>
    signal(mess-disp)
forever
```

Consumatore (C)

```
repeat
    wait(mess-disp)
    <prelievo messaggio>
    <consumazione messaggio>
    signal(spazio-disp)
forever
```

- Cosa succede?

Produttore-Consumatore: esercizi /2/

Produttore (P)

repeat
  wait(spazio-disp)
  wait(mutex)
  <produzione messaggio>
  <deposito messaggio>
  signal(mutex)
  signal(mess-disp)
forever

Consumatore (C)

repeat
  wait(mess-disp)
  wait(mutex)
  <prelievo messaggio>
  <consumazione messaggio>
  signal(mutex)
  signal(spazio-disp)
forever

- Inserito un semaforo mutex con v.i.=1 -- Cosa succede?

Produttore-Consumatore: esercizi /3/

Produttore (P)

```
repeat
    wait(spazio-disp)
    wait(mutex)
    <produzione messaggio>
    <deposito messaggio>
    signal(mutex)
    signal(mess-disp)
forever
```

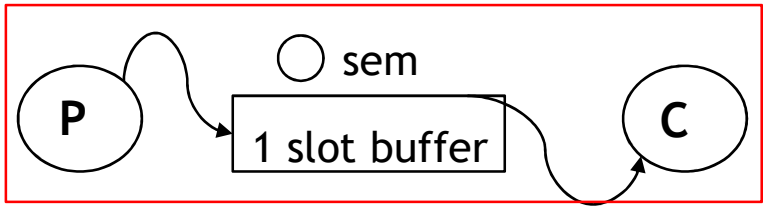
Consumatore (C)

```
repeat
    wait(mess-disp)
    wait(mutex)
    <prelievo messaggio>
    <consumazione messaggio>
    signal(mutex)
    signal(spazio-disp)
forever
```

- E' possibile scambiare l'ordine delle wait nei due processi?

Produttore-Consumatore: esercizi /4/

- Buffer destinato a contenere un solo messaggio (N=1)
- E' possibile semplificare la soluzione utilizzando un solo semaforo sem (v.i. sem=1)?



Produttore (P)

```
repeat
    <produzione messaggio>
    wait(sem)
    <deposito messaggio>
    signal(sem)
forever
```

Consumatore (C)

```
repeat
    wait(sem)
    <prelievo messaggio>
    signal(sem)
    <consumazione messaggio>
forever
```

Produttore-Consumatore: esercizi /5/

- Discutere la seguente versione:

Produttore (P)

```
repeat
    <produzione messaggio>
    wait(mess-disp)
    <deposito messaggio>
    signal(spazio-disp)
forever
```

Consumatore (C)

```
repeat
    wait(mess-disp)
    <prelievo messaggio>
    signal(spazio-disp)
    <consumazione messaggio>
forever
```

Produttore-Consumatore: discussione

- [es. 1,2] Soluzioni diverse ai problemi di cooperazione tra processi possono essere caratterizzate da un diverso *grado di concorrenza* (parallelismo potenziale)
- [es. 4] Il problema Produttore-Consumatore è intrinsecamente di complessità più elevata rispetto alla mutua esclusione: occorrono due semafori anche nel caso di buffer ad 1 solo messaggio
- Aumentando la complessità dei problemi di sincronizzazione occorre un *numero crescente di semafori* [seguiranno altri esempi]
- [es. 3,5] Semafori e primitive di sincronizzazione sono meccanismi *potenti ma di basso livello*. E' facile commettere errori nel loro uso

Produttore-Consumatore: osservazioni

- Un'idea: incapsulare le operazioni di produttore e consumatore in procedure di più alto livello

```
procedure send(x: char)
    wait(spazio-disp);
    buffer[P1] := x;
    P1 := (P1+1) mod N;
    signal(mess-disp);
end
```

```
procedure receive(var x: char);
    wait(mess-disp);
    x := buffer[P2];
    P2 := (P2+1) mod N;
    signal(spazio-disp);
end
```

- Produttore e consumatore potrebbero invocare le procedure *send* e *receive* (assunte come *primitive*) senza dover usare i semafori

Produttore-Consumatore: osservazioni

Produttore (P)

```
var c: char;
repeat
    <produzione messaggio>
    send(c)
    …
forever
```

Consumatore (C)

```
var m: char;
repeat
    receive(m)
    <consumazione messaggio>
    …
forever
```

- Primitive di sincronizzazione del tipo *send* e *receive* per innalzare il livello di astrazione offerto
- (idea da sviluppare… manca modello per designazione!)

Quesito

- Soluzione del problema Produttore-Consumatore:

  mess-disp v.i. = 0, *spazio-disp* v.i. = N

Produttore (Pi)

```
repeat
    <produzione messaggio>
    wait(spazio-disp)
    <deposito messaggio>
    signal(mess-disp)
forever
```

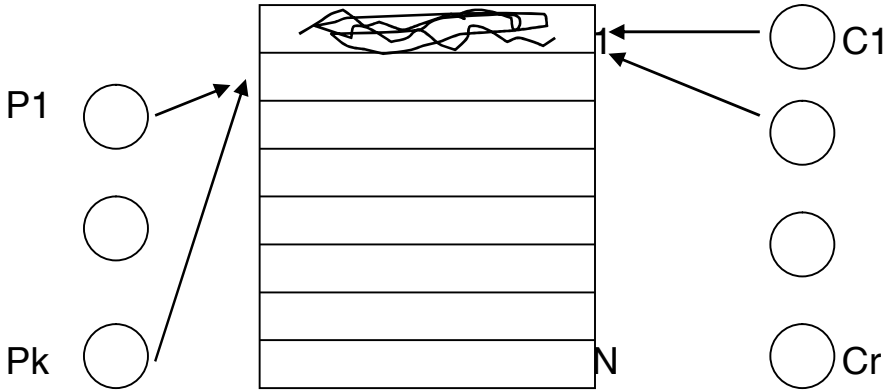
Consumatore (Cj)

```
repeat
    wait(mess-disp)
    <prelievo messaggio>
    signal(spazio-disp)
    <consumazione messaggio>
forever
```

- La soluzione funziona in presenza di *più produttori e/o consumatori*?

Produttori-Consumatori

- Come cambia il problema in presenza di più produttori e/o più consumatori?



- Deposito ora è una sezione critica per i produttori, Prelievo è una sezione critica per i consumatori

Produttori-Consumatori

Produttore (Pi)

```
repeat
    <produzione messaggio>
    wait(spazio-disp)
    wait(mutex1)
    <deposito messaggio>
    signal(mutex1)
    signal(mess-disp)
forever
```

Consumatore (Cj)

```
repeat
    wait(mess-disp)
    wait(mutex2)
    <prelievo messaggio>
    signal(mutex2)
    signal(spazio-disp)
    <consumazione messaggio>
forever
```

- mutex1, mutex2 v.i. 1; mess-disp v.i. 0; spazio-disp v.i. N

Produttori-Consumatori: discussione

- Unico semaforo mutex al posto di mutex1 e mutex2?
  → *deposito* e *prelievo* sono considerate sezioni critiche della stessa classe
- Due semafori distinti mutex1 e mutex2 consentono di parallelizzare le attività di un produttore e un consumatore sul buffer → massimo parallelismo possibile

Produttori-Consumatori: discussione

- Cosa succede se si scambia nei thread l'ordine di accesso alle sezioni critiche?

Produttore (Pi)

```
repeat
    <produzione messaggio>
    wait(mutex1)
    wait(spazio-disp)
    <deposito messaggio>
    signal(mess-disp)
    signal(mutex1)
forever
```

Consumatore (Cj)

```
repeat
    wait(mutex2)
    wait(mess-disp)
    <prelievo messaggio>
    signal(spazio-disp)
    signal(mutex2)
    <consumazione messaggio>
forever
```

Semafori e programmazione concorrente

- I semafori sono uno strumento generale per la programmazione concorrente
- Esempio: *mutua esclusione con insieme di risorse equivalenti*
  - n *risorse* {R1, ..., Rn} tra loro *equivalenti*
  - m thread {T1, ..., Tm} devono operare *in modo esclusivo* su una qualunque risorsa Rj, mediante una tra le operazioni {A, B, ...}
  - Rj.A rappresenta l'operazione A eseguita su Rj

Mutua esclusione con risorse equivalenti

- Ad ogni risorsa Rj viene assegnato un semaforo di mutua esclusione Mj con v.i. = 1

  Thread Ti:

  ```
  ...
  wait(Mj);
  Rj.A;
  signal(Mj);
  ...
  ```

- controindicazioni?

Mutua esclusione con risorse equivalenti

- Ad ogni risorsa Rj viene assegnato un semaforo di mutua esclusione Mj con v.i. = 1

  Thread Ti:

  ...
  wait(Mj);
  Rj.A;
  signal(Mj);
  ...

  - Come decide il generico thread Ti su quale risorsa Rj operare ? come viene scelta la risorsa ?
  - Ti, una volta scelta la risorsa Rj, può rimanere bloccato eseguendo wait(Mj) perchè su Rj sta già operando un altro thread Tk. Ti si blocca su wait(Mj) pur essendo disponibili altre risorse Rh (h≠j)

- controindicazioni?

Mutua esclusione con risorse equivalenti: soluzione

- Si introduce una nuova risorsa G, *gestore* di {R1, ..., Rn}. G è costituita da una *struttura dati* destinata a mantenere lo *stato* delle risorse gestite {R1, ..., Rn}
- Sul gestore si opera tramite *due procedure* Richiesta e Rilascio:

  procedure Richiesta (*var* x: 1..n);

  procedure Rilascio (x: 1..n);
- x rappresenta *l'indice* della risorsa *assegnata* o *rilasciata*
  - Richiesta: Esiste una risorsa Rj disponibile?
  - Rilascio(x): Rx ora è disponibile
- Richiesta e Rilascio devono essere eseguite in *mutua esclusione* → semaforo mutex (v.i. = 1)

Mutua esclusione con risorse equivalenti: soluzione

- Semafori: oltre al `mutex` (v.i. = 1) occorre un semaforo `ris` (v.i. = n) per indicare il *numero di risorse disponibil*i
- Un vettore di variabili booleane Libero[i] registra quali risorse sono in ciascun istante libere (Libero[i] = true) e quali occupate (Libero[i] = false)
- thread Ti:

```
var     j: 1..n;
...
Richiesta(j); //attende su j indice della risorsa assegnata
<uso della risorsa j-ma>
Rilascio(j);
...
```

Mutua esclusione con risorse equivalenti: soluzione

```
var     mutex: semaforo v.i.= 1; ris: semaforo v.i. = n;
        Libero: array [1..n] of Boolean v.i. =true;

procedure Richiesta(var x: 1..n);
  var i: 0..n;
  begin
        wait(ris);
        wait(mutex);
        i := 0;
        repeat i := i + 1;
        until Libero [i];
        x := i;
        Libero [i] := false;
        signal(mutex);
  end
```

```
procedure Rilascio(x: 1..n);
  begin
      wait(mutex);
      Libero [x] := true;
      signal(mutex);
      signal(ris);
end
```

Programmazione concorrente mediante semafori

- Quesito generale: Come «scala» l'uso dei semafori in problemi di sincronizzazione più complessi?
- Pattern di programmazione concorrente possono aiutare, ma occorrerà introdurre anche meccanismi di più alto livello
- I semafori sono stati definiti da qualcuno «l'assembly della concorrenza»!